Automatic Formal Analyses of Cryptographic Protocols

Stephen H. Brackin *
Arca Systems, Inc.
ESC/AXS
Hanscom AFB, MA 01731-2116

Abstract

Cryptographic protocols are short sequences of message exchanges intended to establish secure communication over insecure networks; whether they actually do so is a notoriously subtle question. This paper describes results produced by a software tool for automatically proving desired properties of protocols using an extension of the Gong, Needham, Yahalom (GNY) belief logic, if possible, and showing exactly what goes wrong otherwise. The paper gives analyses of three complicated SPX protocols, analyses that reveal serious vulnerabilities. **Keywords***: Protocols; Authentication; Automatic Analysis; Formal Methods.*

1. Introduction

Cryptographic protocols are short sequences of message exchanges intended to establish secure communication over insecure networks. Some do so. Others, including ones recommended by experts, can be subverted by attacks involving modified, replayed, or mislabeled messages [5]. The basic issues are *authentication* (whether participants know who they are communicating with), and *nondisclosure* (whether information is revealed to those not meant to receive it).

There are two main approaches to preventing protocol failure: attempting to construct possible attacks, using algebraic properties of the algorithms in the protocols; and attempting to construct inferences, using specialized logics based on a notion of "belief", that protocol participants can confidently reach desired conclusions.

*The author wishes to thank Shiu-Kai Chin, Grace Hammonds, Randy Lichota, and Jack Wool for their assistance. This work was supported by Rome Laboratory and Air Force Materiel Command's Electronic Systems Center/Software Center (ESC/AXS), Hanscom AFB, through the Portable, Reusable, Integrated Software Modules (PRISM) program, contracts F19628-92-C-0006 and F19628-92-C-0008.

Attack-construction tools include Millen's Interrogator [13, 15, 14] and Meadows' NRL Protocol Analyzer [10, 11, 12]. Inference-construction approaches include the belief logics developed by Abadi and Tuttle (AT)[1], by Gong, Needham, and Yahalom (GNY) [7, 6], and by Syverson and van Oorschot (SvO) [17].

Attack-construction tools address both authentication and nondisclosure, but suffer from a combinatorial explosion in the number of cases they must consider. Belief-logic tools address only authentication, but do not face a combinatorial explosion, are potentially much faster, and are potentially capable of analyzing large, complicated protocols that the attack-construction tools are incapable of analyzing in a reasonable time.

The tool whose results are presented here, the Automatic Authentication Protocol Analyzer (`AAPA`) uses the belief-logic approach. It automatically proves theorems, about a Higher Order Logic (HOL) formalization of a belief logic extending the GNY logic, using HOL proof tools [8]. For the HOL proof tools, whether a claim is a theorem is determined by *type checking* in a Standard ML (SML) [16] compiler. The correctness of the theorems proved by the `AAPA` thus *does not depend* on the correctness of the `AAPA`; it depends only on the correctness of the HOL tools and the SML compiler, which have been used and analyzed extensively.

The `AAPA` automatically translates between HOL and a simple Interface Specification Language (ISL) [4] for describing protocols; the user need only know ISL, not HOL.

The belief logic used by the `AAPA` grew out of the GNY logic, as adapted by Gong to eliminate impossible protocols [6], but it extends the GNY logic in several ways. These extensions include the following:

- having explicit pairing and conjunction operators;
- describing protocol properties at intermediate protocol stages;

- modeling protocols that use Message Authentication Codes (MACs), i.e., key-dependent hash functions;
- modeling protocols that use key-exchange functions to generate shared secret keys;
- modeling protocols that use hash codes or other computed values as keys;
- modeling protocols that use multiple public-key or symmetric-key encryption functions, multiple hash functions, and multiple key-exchange algorithms.

Several of these extensions are necessary for analyzing the protocols described in this paper, since these protocols use hash codes as keys and make essential use of taking different hashes of the same password. The `AAPA` belief logic is described in [3].

It is also worth emphasizing that the original GNY logic existed only on paper; proofs in it were constructed and checked by hand. The AAPA not only gives a machine implementation of a logic, with highly reliable machine checking of the accuracy of proofs, but also constructs these proofs automatically. The AAPA produced the results given in this paper in a matter of minutes; doing much less, constructing machine-checked proofs, by hand, took months [9].

The rest of this paper analyzes three SPX protocols [18] using the `AAPA`. The paper only gives the basic information needed to understand these analyses; see [3, 2, 4] for complete descriptions of the `AAPA`'s underlying HOL theory, its proof process, and the language ISL.

2. SPX Credentials Initialization

An ISL specification for the Credentials Initialization SPX protocol, taken almost verbatim from [18], follows.

The process created by a new user logging in (`C`) contacts the Login Enrollment Agent Facility (`Leaf`), and sends it `C`'s name, a timestamp, a random nonce, and a hash of `C`'s password, all encrypted with `Leaf`'s public key. `Leaf` contacts a Certificate Distribution Center (`Cdc1`), which sends `Leaf` `C`'s long-term private key encrypted with a *different* hash of `C`'s password, the hash of `C`'s password that `Leaf` should have received, and `C`'s user-ID, all encrypted with `Leaf`'s public key. `Leaf` then sends `C` `C`'s user-ID and `C`'s long-term private key encrypted with the different hash of `C`'s password, all encrypted with the random nonce `C` just provided to `Leaf`. From this, `C` is able to determine its own long-term private and public keys. It then contacts `Cdc1` directly and obtains a certificate for the public key for its Certifying Authority `Ca1`, with `C`'s name, `Ca1`'s name, and the validity interval for this public key, all signed by taking the data's hash with yet another hash algorithm and encrypting the result with `C`'s long-term private key.

The protocol's ISL specification follows standard notation, uses intuitive terminology, and is largely self-explanatory. The following descriptions of ISL constructs will suffice for getting a reasonable understanding of the protocol:

- The `From` construct on initial hash codes and encrypted values allows the `AAPA` to compute a putative source for each such value in the protocol; it uses this to direct its proof process.
- `{x}f(k)` denotes `x` encrypted using function `f` with key `k`.
- `[x](f1,f2)(k)` denotes `x` together with a signature produced by taking the hash of `x` using `f1` and encrypting the result using `f2` with key `k`.
- The `||` operator binds a statement to a data item, as in the "extension" concept in the GNY logic [7]. The protocol assumes that the principal originating this data item will not send it unless this principal is confident that the statement is true.
- ISL accepts and ignores C-style comments.

The protocol's ISL specification follows:

```
DEFINITIONS:
PRINCIPALS: C,Ca1,Cdc1,Leaf;
PUBLIC KEYS: KpC,KpCa1,KpLeaf;
PRIVATE KEYS: KsC,KsCa1,KsLeaf;
SYMMETRIC KEYS: Rn;
OTHER: PwdC,UidC,ValidityKpCa1,Ts;
ENCRYPT FUNCTIONS: Des,Rsa;
HASH FUNCTIONS: H1,H2,H3;
Des WITH ANYKEY HASINVERSE Des WITH ANYKEY;
Rsa WITH KpLeaf HASINVERSE Rsa WITH KsLeaf;
Rsa WITH KsC HASINVERSE Rsa WITH KpC;

INITIALCONDITIONS:
C Received
  Des,H1,H2,H3,Rsa,C,PwdC,Ts,KpLeaf,Rn;
C Believes
  (PublicKey Leaf Rsa KpLeaf;
   SharedSecret C C H2(PwdC) From C);
Cdc1 Received
  Rsa,UidC,KpLeaf,
  H1(PwdC) From C,
```

```
  {KsC}Des(H2(PwdC))                  /* 1 */
   ||(PrivateKey C Rsa KsC) From C,
  [C,Ca1,ValidityKpCa1,KpCa1](H3,Rsa)(KsC)
   ||(PublicKey Ca1 Rsa KpCa1) From C;
Leaf Received Des,Rsa,KsLeaf;

PROTOCOL:
1. C -> Leaf: {C,Ts,Rn,H1(PwdC)}Rsa(KpLeaf);
2. Leaf -> Cdc1: C;
3. Cdc1 -> Leaf:
    {{KsC}Des(H2(PwdC))                /* 2 */
      ||(PrivateKey C Rsa KsC),
     H1(PwdC),
     UidC}Rsa(KpLeaf);
4. Leaf -> C:
    {UidC,
     {KsC}Des(H2(PwdC))               /* 3 */
      ||(PrivateKey C Rsa KsC)}Des(Rn);
5. C -> Cdc1: C;
6. Cdc1 -> C:
    [C,Ca1,ValidityKpCa1,KpCa1](H3,Rsa)(KsC)
     ||(PublicKey Ca1 Rsa KpCa1);

GOALS:
4. C Possesses KpC,KsC;
   C Believes
     (PublicKey C Rsa KpC;
      PrivateKey C Rsa KsC);
6. C Possesses ValidityKpCa1,KpCa1;
   C Believes PublicKey Ca1 Rsa KpCa1;
```

The remainder of this paper will assume that this ISL specification is in a file named `spxinit.isl`. Running the `AAPA` on this file gives the error message:

```
User-goal failure, stage: 4!
Goal statement: C Possesses KpC,KsC;
```

and produces files `spxinit.fail` and `spxinit.prvd` containing ISL descriptions of the failed default goals and proved theorems. One of the theorems is `C Received KsC;` so the problem is with `KpC`; the proof rules embodied in the `AAPA` cannot prove that a public key can be computed from the corresponding private key. The assumption that this can be done is implicit in [18]. The necessary machinery to allow the user to specify that a public key is a function of the corresponding secret key is only partially present in the `AAPA`.

This problem can be easily worked around by replacing `KsC` by `KpC,KsC` and

```
PrivateKey C Rsa KsC
```

by

```
PrivateKey C Rsa KsC; PublicKey C Rsa KpC
```

in the lines marked `/* 1 */`, `/* 2 */`, and `/* 3 */`.

This change causes the `AAPA` to give the error;

```
User-goal failure, stage: 4!
Goal statement:
 C Believes
  (PublicKey C Rsa KpC;PrivateKey C Rsa KsC);
```

The `spxinit.fail` file now shows the failed default goal

```
C Believes
 (C Conveyed
  {KpC,KsC}Des(H2(PwdC))
   ||(PublicKey C Rsa KpC;
      PrivateKey C Rsa KsC));
```

The subgoal the prover is unable to prove is:

```
C Believes (C Recognizes KpC,KsC);
```

The goal asserts that `C` can be confident that the keys (key, actually) encrypted with a hash of `C`'s password really originated with `C`. The problem is that any random value can be decrypted with a hash of `C`'s password to produce another random value; how is `C` to know whether the result is `C`'s secret key, which looks random itself? The failed subgoal says `C` can identify the decrypted data as meaningful.

This might be a problem. In the real protocol, the data represented abstractly here as `KsC` might have some structure, or be encrypted with identifying information, so that a decrypted random value can be recognized as meaningless, but the analysis here raises the question for implementations of the protocol as to whether they make such tests.

This new problem can be solved by putting identifying information, `C`'s name, in with the encrypted key(s), and adding the initial condition

```
C Believes C Recognizes C;
```

i.e., `C` can identify its own name as meaningful.

This change causes the `AAPA` to display the same error message that it displayed before, but `spxinit.fail` shows differences; the top failed default goal is now:

```
[C Believes
  (PublicKey C Rsa KpC;
   PrivateKey C Rsa KsC;
   C Possesses Des,H2(PwdC);
   C Possesses C,KpC,KsC);
 C Believes
  (C Believes
    (PublicKey C Rsa KpC;
```

```
      PrivateKey C Rsa KsC;
      C Possesses Des,H2(PwdC);
      C Possesses C,KpC,KsC))]
```

and its waiting subgoals are

```
C Believes
 (Fresh C,KpC,{KsC}Des(H2(PwdC))
  ||(PublicKey C Rsa KpC;
     PrivateKey C Rsa KsC));
C Believes (Trustworthy C);
```

The two parts of this goal reflect that if the belief logic allows the recipient of a data item to believe the properties that the protocol assumes this data item has, then it also allows this recipient to believe that the originator of this data item also believes these properties. The same hypotheses give both conclusions.

The second of the waiting subgoals, that `C` considers itself trustworthy, is trivial and easy to add as an initial condition. The first subgoal, though, reflects a real limitation in the GNY logic.

Following the GNY logic, the `AAPA`'s belief logic does not allow a protocol participant to believe a statement that the protocol assumes is valid for a data item unless this participant has adequate reason to believe that this data item was created for the current protocol run; otherwise it could be a replay. In the current case, `C`'s encrypted private key was not created for the current protocol run, but it has the properties that the protocol assumes it has. As long as the current `PwdC` is no older than the current `KsC`, there is no way for a replay to give `C` a stale private key. The theory's `Fresh` construct, meaning "created for the current run", needs to be generalized to "fresh enough", meaning "having the expected properties for the current run".

This last problem can be solved by having `C` believe that `PwdC` was created for the current run. It might have been, after all, and this assumption accurately reflects that the critical issue is whether `C`'s password is too old.

Adding the initial conditions

```
C Believes (Fresh PwdC; Trustworthy C)
```

gets the `AAPA` past all the default and user-set goals for stage 4. Now it encounters a problem similar to an earlier one:

```
User-goal failure, stage: 6!
Goal statement:
 C Believes (PublicKey Ca1 Rsa KpCa1);
```

Again, the problem is that data that was not created for the current protocol run — `KpCa1` — has to be believed to have the properties the protocol assumes for it. Adding the initial condition

```
C Believes Fresh ValidityKpCa1;
```

causes the `AAPA` to prove the all the user-set goals.

The analysis of the SPX Credentials Initialization protocol reveals some deficiencies in the GNY-based formal theory underlying the `AAPA`, identifies unstated assumptions in [18], and identifies a potential problem — not checking whether the decrypted private key can be identified as being information of an expected form — that the protocol's implementations might have.

3. SPX Authentication

Although it is more complicated, and roughly 50% more difficult to specify, the SPX Authentication protocol raises only issues similar to those raised by the SPX Credentials Initialization protocol.

In this protocol, a claimant (`C`), already possessing its own long-term public and private keys (`KpC`, `KsC`), a pair of shorter-term session public and private keys (`KspC`, `KssC`), and the public key (`KpCa1`) of its Certifying Authority (`Ca1`), and already believing that all these keys are what they are, contacts and verifies its identity to a verifier (`V`) already possessing its own long-term public and private keys (`KpV`, `KsV`) and the public key (`KpCa2`) of its Certifying Authority (`Ca2`), and already believing that these keys are what they are.

`C` contacts a Certificate Distribution Center (`Cdc1`) to obtain a certificate signed with `Ca1`'s private key giving `V`'s public key. `C` then creates a timestamp (`Ts`) and a random symmetric key (`DesKey`), and sends `V` three things:

- An *authenticator*, consisting of `Ts` and `C`'s channel ID `ChannelIdC`, signed with a DES residue of these values produced with `DesKey`. The DES residue is effectively a key-dependent hash.
- A *ticket*, containing the session public key `KspC`, a validity interval `ValidityKspC` for this key, and `C`'s User-ID `UidC`. The ticket is signed with a hash code produced by `H3` and encrypted with `C`'s long-term secret key `KsC`. The ticket communicates `KspC` and identifies it as being from `C`.
- A *delegator*, consisting of `DesKey` encrypted with `V`'s public key, and signed with a hash code produced by `H3` and encrypted with the session secret key `KssC`. The delegator communicates `DesKey` and indirectly identifies it as being from `C`.

After receiving all these things, `V` contacts a Certificate Distribution Center (`Cdc2`) to obtain a certificate signed with `Ca2`'s private key giving `C`'s public key. `V` then checks all the information from `C`, and sends `C Ts`

encrypted with DesKey to confirm its receipt of DesKey and its intent to use it for subsequent communication.

An ISL specification for this protocol follows, including modifications from its description in [18], similar to those for the Credentials Initialization protocol, needed to have it meet all its user-set goals. These modifications include putting the name C in with DesKey and believing that validity intervals are fresh.

```
DEFINITIONS:
PRINCIPALS: C,Ca1,Ca2,Cdc1,Cdc2,V;
PUBLIC KEYS: KpC,KpCa1,KpCa2,KpV,KspC;
PRIVATE KEYS: KsC,KsCa1,KsCa2,KsV,KssC;
SYMMETRIC KEYS: DesKey,Rn;
OTHER: ChannelIdC,UidC,ValidityKpC,
       ValidityKpV,ValidityKspC,Ts;
ENCRYPT FUNCTIONS: Des,Rsa;
KEYED HASH FUNCTIONS: Hdes;
HASH FUNCTIONS: H3;
Des WITH ANYKEY HASINVERSE Des WITH ANYKEY;
Rsa WITH KsC HASINVERSE Rsa WITH KpC;
Rsa WITH KsCa1 HASINVERSE Rsa WITH KpCa1;
Rsa WITH KsCa2 HASINVERSE Rsa WITH KpCa2;
Rsa WITH KpV HASINVERSE Rsa WITH KsV;
Rsa WITH KssC HASINVERSE Rsa WITH KspC;

INITIALCONDITIONS:
C Received
 Des,H3,Hdes,Rsa,
 Ts,ChannelIdC,UidC,ValidityKspC,
 C,V,DesKey,KpC,KsC,KpCa1,KspC,KssC;
C Believes
 (Fresh Ts;
  Fresh ValidityKpV;
  Fresh DesKey;
  Fresh KspC;
  PublicKey C Rsa KspC;
  PublicKey Ca1 Rsa KpCa1;
  PrivateKey C Rsa KssC;
  C Recognizes Ca1;
  C Recognizes Ts;
  SharedSecret C V DesKey;
  Trustworthy Ca1;
  Trustworthy V);
V Received
 Des,H3,Hdes,Rsa,Ts,C,KpV,KsV,KpCa2;
V Believes
 (Fresh Ts;
  Fresh ValidityKpC;
  Fresh ValidityKspC;
  PrivateKey V Rsa KsV;
  PublicKey V Rsa KpV;
  PublicKey Ca2 Rsa KpCa2;
  V Recognizes C;
  V Recognizes Ca2;
  V Recognizes ValidityKspC;
  Trustworthy C;
  Trustworthy Ca2);
Cdc1 Received
 [Ca1,V,ValidityKpV,KpV](H3,Rsa)(KsCa1)
  ||(PublicKey V Rsa KpV) From Ca1;
Cdc2 Received
 [Ca2,C,ValidityKpC,KpC](H3,Rsa)(KsCa2)
  ||(PublicKey C Rsa KpC) From Ca2;

PROTOCOL:
1. C -> Cdc1: V;
2. Cdc1 -> C:
    [Ca1,V,ValidityKpV,KpV](H3,Rsa)(KsCa1)
     ||(PublicKey V Rsa KpV);
3. C -> V:
    <Ts,ChannelIdC>Hdes(DesKey)
     ||(Fresh DesKey),
    [ValidityKspC,UidC,KspC](H3,Rsa)(KsC)
     ||(PublicKey C Rsa KspC;Fresh KspC),
    [{C,DesKey}Rsa(KpV)](H3,Rsa)(KssC)
     ||(SharedSecret C V DesKey);
4. V -> Cdc2: C;
5. Cdc2 -> V:
    [Ca2,C,ValidityKpC,KpC](H3,Rsa)(KsCa2)
     ||(PublicKey C Rsa KpC);
6. V -> C:
    {Ts}Des(DesKey)
     ||(SharedSecret C V DesKey);

GOALS:
2. C Possesses ValidityKpV,KpV;
   C Believes PublicKey V Rsa KpV;
3. V Possesses
    Ts,ChannelIdC,ValidityKspC,
    UidC,KspC,DesKey;
5. V Possesses ValidityKpC,KpC;
   V Believes
    (PublicKey C Rsa KpC;
     PublicKey C Rsa KspC;
     Fresh KspC; Fresh DesKey;
     C Possesses DesKey;
     SharedSecret C V DesKey;
     C Believes SharedSecret C V DesKey);
6. C Believes
    (V Possesses DesKey;
     V Believes SharedSecret C V DesKey);
```

Although the initial-conditions modifications are similar to those for the Credentials Initialization protocol, these modifications are much more questionable for the Authentication protocol.

In the Authentication case, C and V initially believe

ValidityKpV and ValidityKpC are fresh, and use these "fresh enough" beliefs to believe that each others' public keys are what they are. That is not troubling, but V also uses the dubious belief that ValidityKspC is "fresh enough" to derive the even more dubious belief, initially held by C, that KspC was created for the current run. It then uses this conclusion to derive that DesKey is a shared secret between C and V, and uses this shared-secret belief to derive that DesKey was created by C for the current run.

Since SPX session keys are intended to be used for up to 8 hours [18], KspC is really not "fresh enough"; an attacker having a ticket and the corresponding KssC could use a new DesKey and V's widely-available public key to fake an authentication transfer and by doing so impersonate C.

4. SPX Delegation

The SPX Delegation protocol is very similar to the SPX Authentication protocol. The only difference is that the delegator:

```
[{DesKey}Rsa(KpV)](H3,Rsa)(KssC)
  ||(SharedSecret C V DesKey);
```

changes to the new delegator:

```
{DesKey}Rsa(KpV),
{KssC}Des(DesKey)||(PrivateKey C Rsa KssC);
```

Note that, since a signed message is a pair consisting of a message and a signature, the delegators in both protocols are pairs whose first elements are {DesKey}Rsa(KpV). The second element of the pair changes, as does the property associated with this element.

The goals for the Delegation protocol are also very similar to those for the Authentication protocol. The single new goal

```
V Believes PrivateKey C Rsa KssC;
```

replaces the two final stage-5 shared-secret goals in the Authentication protocol.

It turns out, though, that it is impossible to get the proofs of the desired properties to go through, even with dubious "fresh enough" assumptions like those used in the Authentication protocol. Inserting a C in with the encrypted key in {DesKey}Rsa(KpV) and in

```
{KssC}Des(DesKey)||(PrivateKey C Rsa KssC)
```

helps, so that V is able to recognize the encrypted information as meaningful, but there is never a reasonable justification for claiming that

```
V Believes (SharedSecret V C DesKey);
```

Because of this, the default goals

```
V Believes
 (C Conveyed
     {C,KssC}Des(DesKey)
      ||(PrivateKey C Rsa KssC));
V Believes
 (C Conveyed {C,DesKey}Rsa(KpV));
V Believes
 (C Conveyed
     Hdes(DesKey,Ts,ChannelIdC)
      ||(Fresh DesKey));
```

all fail. As described earlier, it is possible for anyone holding a valid ticket from C and a corresponding KssC to create a DesKey and fake a delegation request from C. Further, anyone to whom C has made a delegation request will have such a ticket and a corresponding KssC, and these items will be valid for up to 8 hours. The SPX Delegation protocol does not prevent, say, bankers from obtaining their customers' medical records.

5. Summary

This paper has described a tool that makes it possible to perform careful formal analyses of authentication properties of cryptographic protocols quickly and easily. For the three SPX protocols described in [18], this tool reveals unstated assumptions and a potential weaknesses in the first protocol, and serious weaknesses in the other two protocols. These weaknesses manifest themselves in one case by requiring dubious initial-conditions assumptions to prove the desired conditions, and in the other case by making it impossible to prove these desired conditions even with the dubious assumptions.

References

[1] M. Abadi and M. Tuttle. A semantics for a logic of authentication. In *Proceedings of the 10th Symposium on Principles of Distributed Computing*, pages 201–216. ACM, August 1991.

[2] S. Brackin. Deciding cryptographic protocol adequacy with HOL: The implementation. To Appear in The 1996 International Conference on Theorem Proving in Higher Order Logics, Turku, Finland, August 1996.

[3] S. Brackin. A HOL extension of GNY for automatically analyzing cryptographic protocols. In *Proceedings of Computer Security Foundations Workshop IX*, County Kerry, Ireland, June 1996. IEEE.

[4] S. Brackin. An interface specification language for cryptographic protocols and its translation into HOL. Submitted to the New Security Paradigms Workshop, Arrowhead, CA, September 1996.

[5] D. Denning and G. Sacco. Timestamps in key distribution protocols. *CACM*, 24(8):533–536, August 1981.

[6] L. Gong. Handling infeasible specifications of cryptographic protocols. In *Proceedings of Computer Security Foundations Workshop IV*, pages 99–102, Franconia NH, June 1991. IEEE.

[7] L. Gong, R. Needham, and R. Yahalom. Reasoning about belief in cryptographic protocols. In *Proceedings of the Symposium on Security and Privacy*, pages 234–248, Oakland, CA, May 1990. IEEE.

[8] M. Gordon and T. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, Cambridge, UK, 1993.

[9] G. Hammonds, R. Lichota, G. Hird, and J. Wool. Command center security — proving software correct. In *Proceedings the 10th Annual Conference on Computer Assurance*, pages 163–173, Gaithersburg, MD, June 1995. NIST.

[10] C. Meadows. Using narrowing in the analysis of key management protocols. In *Proceedings of the Symposium on Security and Privacy*, pages 138–147, Oakland, CA, May 1989. IEEE.

[11] C. Meadows. A system for the specification and analysis of key management protocols. In *Proceedings of the Symposium on Security and Privacy*, pages 182–195, Oakland, CA, May 1991. IEEE.

[12] C. Meadows. Applying formal methods to the analysis of a key management protocol. *J. Computer Security*, 1(1):5–36, 1992.

[13] J. Millen. The interrogator: A tool for cryptographic protocol analysis. In *Proceedings of the Symposium on Security and Privacy*, pages 134–141, Oakland, CA, May 1984. IEEE.

[14] J. Millen. The Interrogator model. In *Proceedings of the Symposium on Security and Privacy*, pages 251–260, Oakland, CA, May 1995. IEEE.

[15] J. Millen, S. Clark, and S. Freedman. The Interrogator: Protocol security analysis. *IEEE Trans. on Software Engineering*, SE-13(2):274–288, February 1987.

[16] L. Paulson. *ML for the Working Programmer*. Cambridge University Press, Cambridge, UK, 1993.

[17] P. Syverson and P. van Oorschot. On unifying some cryptographic protocol logics. In *Proceedings of the Symposium on Security and Privacy*, pages 14–28, Oakland, CA, 1994. IEEE.

[18] J. Tardo and K. Alagappan. SPX: Global authentication using public key certificates. In *Proceedings of the Symposium on Security and Privacy*, pages 232–244, Oakland, CA, 1991. IEEE.